Ordering of constraints involving fold expressions

Document #: P2963R2
Date: 2024-05-17
Programming Language C++
Audience: EWG, CWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

Fold expressions, which syntactically look deceptively like conjunctions/subjections for the purpose of constraint ordering are in fact atomic constraints We propose rules for the normalization and ordering of fold expressions over `&&` and `||`.

Revisions

R2

- Wording improvements following CWG review in Tokyio. Notablywe added a description of how satisfaction is established.
- Clarify that subsumption checking short-circuits. Add a design discussion.
- A fold expression constraint can now only subsume another if they both have the same fold operator. This avoid inconsistent subsumption and checking results in the presence of empty packs.

R1

- Wording improvements: The previous version of this paper incorrectly looked at the size of the packs involved in the fold expressions (as it forced partial ordering to look at template arguments). The current design does not look at the template argument/parameter mapping to establish subsumption of fold expressions.
- A complete implementation of this proposal is available on Compiler Explorer. The implementation section was expanded.
- Add an additional example.

Motivation

This paper is an offshoot of P2841R0 [1] which described the issue with lack of subsumption for fold expressions. This was first observed in a Concept TS issue.

This question comes up ever so often on online boards and various chats.

- [StackOverflow] How are fold expressions used in the partial ordering of constraints?
- [StackOverflow] How to implement the generalized form of std::same_as?

In Urbana, core observed ”We can’t constrain variadic templates without fold-expressions” and almost folded (!) fold expressions into the concept TS. The expectation that these features should interoperate well then appear long-standing.

Subsumption and fold expressions over `&&` and `||`

Consider:

```
template <class T> concept A = std::is_move_constructible_v<T>;
template <class T> concept B = std::is_copy_constructible_v<T>;
template <class T> concept C = A<T> && B<T>;

template <class... T>
requires (A<T> && ...)
void g(T...);

template <class... T>
requires (C<T> && ...)
void g(T...);
```

We want to apply the subsumption rule to the normalized form of the requires clause (and its arguments). As of C++23, the above `g` is ambiguous.

This is useful when dealing with algebraic-type classes. Consider a concept constraining a (simplified) environment implementation via a type-indexed `std::tuple`. (In real code, the environment is a type-tag indexed map.)

```
template <typename X, typename... T>
concept environment_of = (... && requires (X& x) { { get<T>(x) } -> std::same_as<T&>; } );

auto f(sender auto&& s, environment_of<std::stop_token> auto env); // uses std::allocator
auto f(sender auto&& s, environment_of<std::stop_token, std::pmr::allocator> auto env); //
    uses given allocator
```

Without the subsumption fixes to fold expressions, the above two overloads conflict, even though they should be partially ordered.

A similar example courtesy of Barry Revzin:

```
template <std::ranges::bidirectional_range R> void f(R&&); // #1
template <std::ranges::random_access_range R> void f(R&&); // #2

template <std::ranges::bidirectional_range... R> void g(R&&...); // #3
template <std::ranges::random_access_range... R> void g(R&&...); // #4
```

C++23	This Paper

`f(std::vector{1, 2, 3}); // Ok`
`g(std::vector{1, 2, 3}); // Error: call to 'g' is ambiguous`
	`f(std::vector{1, 2, 3}); // Ok, calls #2`
	`g(std::vector{1, 2, 3}); // Ok, calls #4`

[Compiler Explorer Demo]

Impact on the standard

This change makes ambiguous overload valid and should not break existing valid code.

Implementabiliy

This was implemented in Clang. Importantly, what we propose does not affect compilers' ability to partially order functions by constraints without instantiating them, nor does it affect the caching of subsumption, which is important to minimize the cost of concepts on compile time: The template arguments of the constraint expressions do not need to be observed to establish subsumption.

An implementation does need to track whether an atomic constraint that contains an unexpanded pack was originally part of a and/or fold expression to properly implement the subsumption rules (`&&` subsumes `||` & `&&` and `||` subsumes `||`).

Subsection with mixed fold operators

Consider this example provided by Hubert Tong

```
template <typename ...T> struct Tuple { };
template <typename T> concept P = true;

template <typename T, typename U, typename V, typename X> struct A;


template <typename ...T, typename ...U, typename V, typename X>
requires P<X> || ((P<V> || P<U>) || ...)          // #A
void foo(A<Tuple<T ...>, Tuple<U ...>, V, X> *); // #1

template <typename ...T, typename ...U, typename V, typename X>
requires P<X> || ((P<V> && P<T>) && ...)          // #B
void foo(A<Tuple<T ...>, Tuple<U ...>, V, X> *); // #2


void bar(A<Tuple<int>, Tuple<>, int, int> *p) { foo(p); }
```

In this example, under the rule proposed in R1, of this paper, #A subsumed #B, and so #1 would have been be a better choice. However here, `U` is empty. So A's constraints are equivalent to just `P<X>` which make B more constrained.

To avoid inconsistencies between constraint checking and subsumption, a fold expression can only subsume another if they both have the same fold operator (they are both folding over `&&` or they are both folding over `||`).

Short circuiting

To be consistent with conjunction constraint and disjection constraints, we propose that fold expanded constrait short circuit (both their evaluation and substitution).

What this paper is not

When the pattern of the `fold-expressions` is a 'concept' template parameter, this paper does not apply. In that case, we need different rules which are covered in P2841R0 [1] along with the rest of the "concept template parameter" feature (specifically, for concept patterns we need to decompose each concept into its constituent atomic constraints and produce a fully decomposed sequence of conjunction/disjunction)

Design and wording strategy

To simplify the wording, we first normalize fold expressions to extract the non-pack expression of binary folds into its own normalized form, and transform `(... && A)` into `(A && ...)` as they are semantically identical for the purpose of subsumption. We then are left with either `(A && ...)` or `(A || ...)`, and for packs of the same size, the rules of subsumptions are the same as for that of atomic constraints.

Wording

� Constraints [temp.constr.constr]

� General [temp.constr.constr.general]

A *constraint* is a sequence of logical operations and operands that specifies requirements on template arguments. The operands of a logical operation are constraints. There are ~~three~~ four different kinds of constraints:

- conjunctions [temp.constr.op],
- disjunctions [temp.constr.op], ~~and~~
- atomic constraints [temp.constr.atomic], and
- fold expanded constraints [temp.constr.fold].

In order for a constrained template to be instantiated [temp.spec], its associated constraints [temp.constr.decl] shall be satisfied as described in the following subclauses. [*Note:* Forming the name of a specialization of a class template, a variable template, or an alias template [temp.names] requires the satisfaction of its constraints. Overload resolution [over.match.viable] requires the satisfaction of constraints on functions and function templates. —*end note*]

� Logical operations [temp.constr.op]

There are two binary logical operations on constraints: conjunction and disjunction. [*Note:* These logical operations have no corresponding C++ syntax. For the purpose of exposition, conjunction is spelled using the symbol $\wedge$ and disjunction is spelled using the symbol $\vee$. The operands of these operations are called the left and right operands. In the constraint $A \wedge B$, A is the left operand, and B is the right operand. —*end note*]

A *conjunction* is a constraint taking two operands. To determine if a conjunction is *satisfied*, the satisfaction of the first operand is checked. If that is not satisfied, the conjunction is not satisfied. Otherwise, the conjunction is satisfied if and only if the second operand is satisfied.

A *disjunction* is a constraint taking two operands. To determine if a disjunction is *satisfied*, the satisfaction of the first operand is checked. If that is satisfied, the disjunction is satisfied. Otherwise, the disjunction is satisfied if and only if the second operand is satisfied.

[*Example:*

```
template<typename T>
constexpr bool get_value() { return T::value; }

template<typename T>
requires (sizeof(T) > 1) && (get_value<T>())
void f(T);      // has associated constraint sizeof(T) > 1 ∧ get_value<T>()

void f(int);

f('a'); // OK, calls f(int)
```

In the satisfaction of the associated constraints[temp.constr.decl] of `f`, the constraint `sizeof(char) > 1` is not satisfied; the second operand is not checked for satisfaction. —*end example*]

[*Note:* A logical negation expression [expr.unary.op] is an atomic constraint; the negation operator is not treated as a logical operation on constraints. As a result, distinct negation *constraint-expression*s that are equivalent under **??** do not subsume one another under **??**. Furthermore, if substitution to determine whether an atomic constraint is satisfied [temp.constr.atomic] encounters a substitution failure, the constraint is not satisfied, regardless of the presence of a negation operator. [*Example:*

```
template <class T> concept sad = false;

template <class T> int f1(T) requires (!sad<T>);
template <class T> int f1(T) requires (!sad<T>) && true;
int i1 = f1(42);        // ambiguous, !sad<T> atomic constraint expressions
    [temp.constr.atomic]
// are not formed from the same expression

template <class T> concept not_sad = !sad<T>;
template <class T> int f2(T) requires not_sad<T>;
template <class T> int f2(T) requires not_sad<T> && true;
int i2 = f2(42);        // OK, !sad<T> atomic constraint expressions both come from not_sad

template <class T> int f3(T) requires (!sad<typename T::type>);
int i3 = f3(42);        // error: associated constraints not satisfied due to substitution
    failure

template <class T> concept sad_nested_type = sad<typename T::type>;
template <class T> int f4(T) requires (!sad_nested_type<T>);
int i4 = f4(42);        // OK, substitution failure contained within sad_nested_type
```

Here, `requires (!sad<typename T::type>)` requires that there is a nested `type` that is not `sad`, whereas `requires (!sad_nested_type<T>)` requires that there is no `sad` nested `type`. *— end example*] *— end note*]

� Atomic constraints [temp.constr.atomic]

An *atomic constraint* is formed from an expression `E` and a mapping from the template parameters that appear within `E` to template arguments that are formed via substitution during constraint normalization in the declaration of a constrained entity (and, therefore, can involve the unsubstituted template parameters of the constrained entity), called the *parameter mapping* [temp.constr.decl]. [*Note:* Atomic constraints are formed by constraint normalization[temp.constr.normal]. `E` is never a logical and expression**??** nor a logical or expression**??**. *— end note*]

Two atomic constraints, e_1 and e_2, are *identical* if they are formed from the same appearance of the same *expression* and if, given a hypothetical template A whose *template-parameter-list* consists of *template-parameter* s corresponding and equivalent[temp.over.link] to those mapped by the parameter mappings of the expression, a *template-id* naming A whose *template-argument* s are the targets of the parameter mapping of e_1 is the same[temp.type] as a *template-id* naming A whose *template-argument* s are the targets of the parameter mapping of e_2. [*Note:* The comparison of parameter mappings of atomic constraints operates in a manner similar to that of declaration matching with alias template substitution[temp.alias]. [*Example:*

```
template <unsigned N> constexpr bool Atomic = true;
template <unsigned N> concept C = Atomic<N>;
template <unsigned N> concept Add1 = C<N + 1>;
template <unsigned N> concept AddOne = C<N + 1>;
template <unsigned M> void f()
requires Add1<2 * M>;
template <unsigned M> int f()
```

```
        requires AddOne<2 * M> && true;

        int x = f<0>();     // OK, the atomic constraints from concept C in both fs are
Atomic<N>
        // with mapping similar to N ↦ 2 * M + 1

        template <unsigned N> struct WrapN;
        template <unsigned N> using Add1Ty = WrapN<N + 1>;
        template <unsigned N> using AddOneTy = WrapN<N + 1>;
        template <unsigned M> void g(Add1Ty<2 * M> *);
        template <unsigned M> void g(AddOneTy<2 * M> *);

        void h() {
            g<0>(nullptr);    // OK, there is only one g
        }
```

— *end example*] As specified in **??**, if the validity or meaning of the program depends on whether two constructs are equivalent, and they are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required. [*Example:*

```
        template <unsigned N> void f2()
        requires Add1<2 * N>;
        template <unsigned N> int f2()
        requires Add1<N * 2> && true;
        void h2() {
            f2<0>();          // ill-formed, no diagnostic required:
            // requires determination of subsumption between atomic constraints that are
            // functionally equivalent but not equivalent
        }
```

— *end example*] — *end note*]

To determine if an atomic constraint is *satisfied*, the parameter mapping and template arguments are first substituted into its expression. If substitution results in an invalid type or expression in the immediate context of the atomic constraint[temp.deduct.general], the constraint is not satisfied. Otherwise, the lvalue-to-rvalue conversion[conv.lval] is performed if necessary, and `E` shall be a constant expression of type `bool`. The constraint is satisfied if and only if evaluation of `E` results in `true`. If, at different points in the program, the satisfaction result is different for identical atomic constraints and template arguments, the program is ill-formed, no diagnostic required. [*Example:*

```
    template<typename T> concept C =
    sizeof(T) == 4 && !true;      // requires atomic constraints sizeof(T) == 4 and !true

    template<typename T> struct S {
        constexpr operator bool() const { return true; }
    };

    template<typename T> requires (S<T>{})
    void f(T);                      // #1
    void f(int);                    // #2

    void g() {
        f(0);                         // error: expression S<int>{} does not have type
bool
    }                               // while checking satisfaction of deduced arguments
of #1;
    // call is ill-formed even though #2 is a better match
```

— *end example*]

� Fold expanded constraint [temp.constr.fold]

A *fold expanded constraint* is formed from a constraint and a *fold-operator* which can either be `&&` or `||`.

A fold expanded constraint whose *fold-operator* is `&&` is a *fold expanded conjunction constraint*.

A fold expanded constraint whose *fold-operator* is `||` is a *fold expanded disjunction constraint*.

A fold expanded constraint is a pack expansion. Let N be the number of elements in the pack expansion that would result from its instantiation [temp.variadic].

A fold expanded conjunction constraint is satisfied if N equals 0 or if for each $0 \leq i < N$ in increasing order, its constraint is satisfied when replacing each pack expansion parameter with the corresponding ith element. No substitution takes place for any i greater than the smallest i for which the constraint is not satisfied.

A fold expanded disjunction constraint is satisfied, if N is not 0 and for any $0 \leq i < N$ in increasing order its constraint is satisfied when replacing each pack expansion parameter with the corresponding ith element. No substitution takes place for any i greater than the smallest i for which the constraint is satisfied.

[*Editor's note:* [...]]

� Constraint normalization [temp.constr.normal]

The *normal form* of an *expression* `E` is a constraint[temp.constr.constr] that is defined as follows:

- The normal form of an expression `( E )` is the normal form of `E`.
- The normal form of an expression `E1 || E2` is the disjunction[temp.constr.op] of the normal forms of `E1` and `E2`.
- The normal form of an expression `E1 && E2` is the conjunction of the normal forms of `E1` and `E2`.
- The normal form of a concept-id `C<`A_1`,` A_2`, ...,` A_n`>` is the normal form of the *constraint-expression* of `C`, after substituting A_1, A_2, ..., A_n for `C`'s respective template parameters

in the parameter mappings in each atomic constraint. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required. [*Example:*

```
template<typename T> concept A = T::value || true;
template<typename U> concept B = A<U*>;
template<typename V> concept C = B<V&>;
```

Normalization of `B`'s *constraint-expression* is valid and results in `T::value` (with the mapping `T` $\mapsto$ `U*`) $\vee$ `true` (with an empty mapping), despite the expression `T::value` being ill-formed for a pointer type `T`. Normalization of `C`'s *constraint-expression* results in the program being ill-formed, because it would form the invalid type `V&*` in the parameter mapping. — *end example*]

- For a *fold-operator* [expr.prim.fold] that is either `&&` or `||`, the normal form of an expression `( ... `*`fold-operator`*` E )` is the normal form of `( E `*`fold-operator`*`...)`.
- The normal form of an expression `( E1 `*`fold-operator`*` ... `*`fold-operator`*` E2 )` is the the normal form of
  - `(E1 `*`fold-operator`*`...) `*`fold-operator`*` E2` if `E1` contains an unexpanded pack, or
  - `E1 `*`fold-operator`*` (E2 `*`fold-operator`*`...)` otherwise.
- The normal form of `(E && ...)` is a fold expanded conjunction constraint [temp.constr.fold] whose constraint is the normal form of `E`.
- The normal form of `(E || ...)` is a fold expanded disjunction constraint whose constraint is the normal form of `E`.
- The normal form of any other expression `E` is the atomic constraint whose expression is `E` and whose parameter mapping is the identity mapping.

� Partial ordering by constraints [temp.constr.order]

A constraint P *subsumes* a constraint Q if and only if, for every disjunctive clause P_i in the disjunctive normal form of P, P_i subsumes every conjunctive clause Q_j in the conjunctive normal form of Q, where

- a disjunctive clause P_i subsumes a conjunctive clause Q_j if and only if there exists an atomic constraint P_{ia} in P_i for which there exists an atomic constraint Q_{jb} in Q_j such that P_{ia} subsumes Q_{jb}, and
- an atomic constraint A subsumes another atomic constraint B if and only if A and B are identical using the rules described in [temp.constr.atomic].
- a fold expanded constraint A subsumes another fold expanded constraint B if both A and B have the same *fold-operator* and the constraint of A subsumes that of B.

[*Example:* Let A and B be atomic constraints [temp.constr.atomic]. The constraint $A \wedge B$ subsumes A, but A does not subsume $A \wedge B$. The constraint A subsumes $A \vee B$, but $A \vee B$ does not subsume A. Also note that every constraint subsumes itself. — *end example*]

Acknowledgments

Thanks to Robert Haberlach for creating the original Concept TS issue.

Thanks to Jens Mauer and Barry Revzin for their input on the wording.

References

[1] Corentin Jabot and Gašper Ažman. P2841R0: Concept template parameters. https://wg21.link/p2841r0, 5 2023.

[N4958] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4958